

Kinds Are Calling Conventions

**Paul Downen, Zena M. Ariola,
Simon Peyton Jones, Richard A. Eisenberg**

PLDI 2022, June 15—17
Originally in ICFP 2020, August 23—29

Efficient Function Calls

Parameter Passing Techniques

Efficient Function Calls

Parameter Passing Techniques

- Representation — What & Where?

Efficient Function Calls

Parameter Passing Techniques

- Representation — What & Where?
- **Arity** — How many?

Efficient Function Calls

Parameter Passing Techniques

- Representation — What & Where?
- Arity — How many?
- **Levity (aka Evaluation Strategy) — When to compute?**

Efficient Function Calls

Parameter Passing Techniques

- Representation — What & Where?
- **Arity** — How many?
- Levity (aka Evaluation Strategy) — When to compute?

Determining Function Arity

`f1, f2, f3, f4 :: Int -> Int -> Int`

Type suggests arity 2

Determining Function Arity

`f1, f2, f3, f4 :: Int -> Int -> Int`

Type suggests arity 2

```
f1 = \x -> \y ->
  let z = expensive x
  in y + z
```


Determining Function Arity

`f1, f2, f3, f4 :: Int -> Int -> Int`

Type suggests arity 2

`f1 = \x -> \y ->`

`let z = expensive x` **Arity 2**

`in y + z`

Determining Function Arity

`f1, f2, f3, f4 :: Int -> Int -> Int`

Type suggests arity 2

`f1 = \x -> \y ->`

`let z = expensive x` **Arity 2** `f2 = \x -> f1 x`
`in y + z`

Determining Function Arity

`f1, f2, f3, f4 :: Int -> Int -> Int`

Type suggests arity 2

`f1 = \x -> \y ->`

`let z = expensive x`
`in y + z`

Arity 2

`f2 = \x -> f1 x`

`= \x -> \y -> f1 x y`

Determining Function Arity

`f1, f2, f3, f4 :: Int -> Int -> Int`

Type suggests arity 2

`f1 = \x -> \y ->`

`let z = expensive x` **Arity 2**
`in y + z`

`f2 = \x -> f1 x` **Arity 2**
`= \x -> \y -> f1 x y`

Determining Function Arity

`f1, f2, f3, f4 :: Int -> Int -> Int`

Type suggests arity 2

`f1 = \x -> \y ->`

`let z = expensive x` **Arity 2**
`in y + z`

`f2 = \x -> f1 x` **Arity 2**
`= \x -> \y -> f1 x y`

`f3 = \x ->`

`let z = expensive x`
`in \y -> y + z`

Determining Function Arity

`f1, f2, f3, f4 :: Int -> Int -> Int`

Type suggests arity 2

`f1 = \x -> \y ->`

`let z = expensive x` **Arity 2**
`in y + z`

`f2 = \x -> f1 x` **Arity 2**
`= \x -> \y -> f1 x y`

`f3 = \x ->`

`let z = expensive x`
`in \y -> y + z`

Hint: 'expensive x' may be costly, or even cause side effects

Determining Function Arity

`f1, f2, f3, f4 :: Int -> Int -> Int` Type suggests arity 2

`f1 = \x -> \y ->`

`let z = expensive x Arity 2 f2 = \x -> f1 x Arity 2`
`in y + z = \x -> \y -> f1 x y`

`f3 = \x ->`

`let z = expensive x Arity 1`
`in \y -> y + z`

Hint: 'expensive x' may be costly, or even cause side effects

Determining Function Arity

`f1, f2, f3, f4 :: Int -> Int -> Int` Type suggests arity 2

`f1 = \x -> \y ->`

`let z = expensive x Arity 2 f2 = \x -> f1 x Arity 2`
`in y + z = \x -> \y -> f1 x y`

`f3 = \x ->`

`let z = expensive x Arity 1 f4 = \x -> f3 x`
`in \y -> y + z`

Hint: 'expensive x' may be costly, or even cause side effects

What Is Arity?

For Curried Functions

What Is Arity?

For Curried Functions

Definition 1. The number of arguments a function needs to do “serious work.”

What Is Arity?

For Curried Functions

Definition 1. The number of arguments a function needs to do “serious work.”

- If ‘ $f\ x\ y\ z$ ’ does work, but ‘ $f\ x\ y$ ’ does not, then ‘ f ’ has arity 3

What Is Arity?

For Curried Functions

Definition 1. The number of arguments a function needs to do “serious work.”

- If ‘ $f\ x\ y\ z$ ’ does work, but ‘ $f\ x\ y$ ’ does not, then ‘ f ’ has arity 3

Definition 2. The number of times a function may be η -expanded without changing its behavior or cost.

What Is Arity?

For Curried Functions

Definition 1. The number of arguments a function needs to do “serious work.”

- If ‘ $f\ x\ y\ z$ ’ does work, but ‘ $f\ x\ y$ ’ does not, then ‘ f ’ has arity 3

Definition 2. The number of times a function may be η -expanded without changing its behavior or cost.

- If ‘ f ’ is equivalent to ‘ $\lambda x\ y\ z.\ f\ x\ y\ z$ ’, then ‘ f ’ has arity 3

What Is Arity?

For Curried Functions

Definition 1. The number of arguments a function needs to do “serious work.”

- If ‘ $f\ x\ y\ z$ ’ does work, but ‘ $f\ x\ y$ ’ does not, then ‘ f ’ has arity 3

Definition 2. The number of times a function may be η -expanded without changing its behavior or cost.

- If ‘ f ’ is equivalent to ‘ $\lambda x\ y\ z.\ f\ x\ y\ z$ ’, then ‘ f ’ has arity 3

Definition 3. The number of arguments passed simultaneously to a function during one call.

What Is Arity?

For Curried Functions

Definition 1. The number of arguments a function needs to do “serious work.”

- If ‘ $f\ x\ y\ z$ ’ does work, but ‘ $f\ x\ y$ ’ does not, then ‘ f ’ has arity 3

Definition 2. The number of times a function may be η -expanded without changing its behavior or cost.

- If ‘ f ’ is equivalent to ‘ $\lambda x\ y\ z.\ f\ x\ y\ z$ ’, then ‘ f ’ has arity 3

Definition 3. The number of arguments passed simultaneously to a function during one call.

- If ‘ f ’ has arity 3, then ‘ $f\ x\ y\ z$ ’ can be implemented as a single call

What Is Arity?

For Curried Functions

Definition 1. The number of arguments a function needs to do “serious work.”

- If ‘ $f\ x\ y\ z$ ’ does work, but ‘ $f\ x\ y$ ’ does not, then ‘ f ’ has arity 3

Definition 2. The number of times a function may be η -expanded without changing its behavior or cost.

- If ‘ f ’ is equivalent to ‘ $\lambda x\ y\ z.\ f\ x\ y\ z$ ’, then ‘ f ’ has arity 3

Definition 3. The number of arguments passed simultaneously to a function during one call.

- If ‘ f ’ has arity 3, then ‘ $f\ x\ y\ z$ ’ can be implemented as a single call

Goal: An IL with *unrestricted* η
for functions, along with
restricted β for other types

Static Arity

In an Intermediate Language

Static Arity

In an Intermediate Language

- New $a \rightsquigarrow b$ type of primitive functions (ASCII 'a ~> b')
 - To distinguish from the source-level $a \rightarrow b$ with different semantics

Static Arity

In an Intermediate Language

- New $a \rightsquigarrow b$ type of primitive functions (ASCII ‘a ~> b’)
 - To distinguish from the source-level $a \rightarrow b$ with different semantics
- Primitive functions are *fully extensional*, **unlike** source functions
 - $\lambda x.f\ x =_{\eta} f : a \rightsquigarrow b$ *unconditionally*

Static Arity

In an Intermediate Language

- New $a \rightsquigarrow b$ type of primitive functions (ASCII ‘a ~> b’)
 - To distinguish from the source-level $a \rightarrow b$ with different semantics
- Primitive functions are *fully extensional*, **unlike** source functions
 - $\lambda x . f x =_{\eta} f : a \rightsquigarrow b$ *unconditionally*
- Application may still be *restricted* for efficiency, like source functions
 - $(\lambda x . x + x)$ (*expensive* 10^6) does not recompute *expensive* 10^6

Static Arity

In an Intermediate Language

- New $a \rightsquigarrow b$ type of primitive functions (ASCII ‘a ~> b’)
 - To distinguish from the source-level $a \rightarrow b$ with different semantics
- Primitive functions are *fully extensional*, **unlike** source functions
 - $\lambda x . f x =_{\eta} f : a \rightsquigarrow b$ *unconditionally*
- Application may still be *restricted* for efficiency, like source functions
 - $(\lambda x . x + x)$ (*expensive* 10^6) does not recompute *expensive* 10^6
- With full η , types express arity — just count the arrows
 - $f : Int \rightsquigarrow Bool \rightsquigarrow String$ has arity 2, no matter f 's definition

Currying

When Partial Application Matters

Currying

When Partial Application Matters

`f3 :: Int -> Int -> Int`

`f3 = \x -> let z = expensive x in \y -> y + z`

Currying

When Partial Application Matters

`f3 :: Int -> Int -> Int`

`f3 = \x -> let z = expensive x in \y -> y + z`

- Because of η , `f3` now has arity 2, not 1!

Currying

When Partial Application Matters

```
f3 :: Int -> Int -> Int
```

```
f3 = \x -> let z = expensive x in \y -> y + z
```

- Because of η , `f3` now has arity 2, not 1!
 - `map (f3 100) [1..106]` recomputes 'expensive 100' a million times 😞

Currying

When Partial Application Matters

```
f3 :: Int ~> Int ~> Int
```

```
f3 = \x -> let z = expensive x in \y -> y + z
```

- Because of η , `f3` now has arity 2, not 1!
 - `map (f3 100) [1..106]` recomputes 'expensive 100' a million times 😞

```
f3' :: Int ~> { Int ~> Int }
```

```
f3' = \x -> let z = expensive x in Clos (\y -> y + z)
```

```
Clos :: (Int ~> Int) ~> {Int ~> Int}
```

Currying

When Partial Application Matters

$f3 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

$f3 = \lambda x \rightarrow \text{let } z = \text{expensive } x \text{ in } \lambda y \rightarrow y + z$

- Because of η , $f3$ now has arity 2, not 1!
 - $\text{map } (f3 \ 100) [1..10^6]$ recomputes 'expensive 100' a million times ☹️

$f3' :: \text{Int} \rightarrow \{ \text{Int} \rightarrow \text{Int} \}$

$f3' = \lambda x \rightarrow \text{let } z = \text{expensive } x \text{ in } \text{Clos } (\lambda y \rightarrow y + z)$

- $f3'$ is an arity 1 function; returns a closure $\{\text{Int} \rightarrow \text{Int}\}$ of an arity 1 function

$\text{Clos} :: (\text{Int} \rightarrow \text{Int}) \rightarrow \{\text{Int} \rightarrow \text{Int}\}$

Currying

When Partial Application Matters

$f3 :: \text{Int} \sim> \text{Int} \sim> \text{Int}$

$f3 = \backslash x \rightarrow \text{let } z = \text{expensive } x \text{ in } \backslash y \rightarrow y + z$

- Because of η , $f3$ now has arity 2, not 1!
 - $\text{map } (f3 \ 100) [1..10^6]$ recomputes 'expensive 100' a million times ☹️

$f3' :: \text{Int} \sim> \{ \text{Int} \sim> \text{Int} \}$

$f3' = \backslash x \rightarrow \text{let } z = \text{expensive } x \text{ in } \text{Clos } (\backslash y \rightarrow y + z)$

- $f3'$ is an arity 1 function; returns a closure $\{\text{Int} \sim> \text{Int}\}$ of an arity 1 function
 - $\text{map } (\text{App } (f3' \ 100)) [1..10^6]$ computes 'expensive 100' only once 😊

$\text{Clos} :: (\text{Int} \sim> \text{Int}) \sim> \{\text{Int} \sim> \text{Int}\}$ $\text{App} :: \{\text{Int} \sim> \text{Int}\} \sim> \text{Int} \sim> \text{Int}$

Functions are Called

Not Evaluated

Functions are Called

Not Evaluated

`x = let f :: Int -> Int = expensive 100 in ...f...f...`

Functions are Called

Not Evaluated

`x = let f :: Int ~> Int = expensive 100 in ...f...f...`

- When is `expensive 100` evaluated?

Functions are Called

Not Evaluated

$x = \text{let } f :: \text{Int} \sim> \text{Int} = \text{expensive } 100 \text{ in } \dots f \dots f \dots$

- When is `expensive 100` evaluated?
 - Call-by-value: first, before binding `f`

Functions are Called

Not Evaluated

$x = \text{let } f :: \text{Int} \sim> \text{Int} = \text{expensive } 100 \text{ in } \dots f \dots f \dots$

- When is `expensive 100` evaluated?
 - Call-by-value: first, before binding `f`
 - Call-by-need: later, but only once, when `f` is first demanded

Functions are Called

Not Evaluated

$x = \text{let } f :: \text{Int} \sim> \text{Int} = \text{expensive } 100 \text{ in } \dots f \dots f \dots$

- When is `expensive 100` evaluated?
 - Call-by-value: first, before binding `f`
 - Call-by-need: later, but only once, when `f` is first demanded
 - Call-by-name: later, re-evaluated every time `f` is demanded

Functions are Called

Not Evaluated

$x = \text{let } f :: \text{Int} \sim> \text{Int} = \text{expensive } 100 \text{ in } \dots f \dots f \dots$

- When is `expensive 100` evaluated?
 - Call-by-value: first, before binding `f`
 - Call-by-need: later, but only once, when `f` is first demanded
 - Call-by-name: later, re-evaluated every time `f` is demanded

$x' = \text{let } f :: \text{Int} \sim> \text{Int} = \lambda y \rightarrow \text{expensive } 100 \ y \text{ in } \dots f \dots f \dots$

Functions are Called

Not Evaluated

$x = \text{let } f :: \text{Int} \sim> \text{Int} = \text{expensive } 100 \text{ in } \dots f \dots f \dots$

- When is `expensive 100` evaluated?
 - Call-by-value: first, before binding `f`
 - Call-by-need: later, but only once, when `f` is first demanded
 - Call-by-name: later, re-evaluated every time `f` is demanded

$x' = \text{let } f :: \text{Int} \sim> \text{Int} = \lambda y \rightarrow \text{expensive } 100 \ y \text{ in } \dots f \dots f \dots$

- $x = x'$ by η , and x' always follows call-by-name order!

Functions are Called

Not Evaluated

$x = \text{let } f :: \text{Int} \sim> \text{Int} = \text{expensive } 100 \text{ in } \dots f \dots f \dots$

- When is `expensive 100` evaluated?
 - Call-by-value: first, before binding `f`
 - Call-by-need: later, but only once, when `f` is first demanded
 - Call-by-name: later, re-evaluated every time `f` is demanded

$x' = \text{let } f :: \text{Int} \sim> \text{Int} = \lambda y \rightarrow \text{expensive } 100 \ y \text{ in } \dots f \dots f \dots$

- $x = x'$ by η , and x' always follows call-by-name order!
- Primitive functions are never just *evaluated*; they are always *called*

The Problem With Polymorphism

And Static Compilation

The Problem With Polymorphism

And Static Compilation

```
poly :: forall a. (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 5, g 4)
```

The Problem With Polymorphism

And Static Compilation

$\text{poly} :: \text{forall } a. (\text{Int} \rightarrow \text{Int} \rightarrow a) \rightarrow (a, a)$
 $\text{poly } f = \text{let } g :: \text{Int} \rightarrow a = f \ 3 \ \text{in } (g \ 5, g \ 4)$

- What are the arities of f and g ? Counting arrows...

The Problem With Polymorphism

And Static Compilation

$\text{poly} :: \text{forall } a. (\text{Int} \sim> \text{Int} \sim> a) \sim> (a, a)$
 $\text{poly } f = \text{let } g :: \text{Int} \sim> a = f \ 3 \ \text{in } (g \ 5, g \ 4)$

- What are the arities of f and g ? Counting arrows...
 - $f :: \text{Int} \sim> \text{Int} \sim> a$ has arity 2

The Problem With Polymorphism

And Static Compilation

`poly :: forall a. (Int ~> Int ~> a) ~> (a, a)`
`poly f = let g :: Int ~> a = f 3 in (g 5, g 4)`

- What are the arities of `f` and `g`? Counting arrows...
 - `f :: Int ~> Int ~> a` has arity 2
 - `g :: Int ~> a` has arity 1

The Problem With Polymorphism

And Static Compilation

```
poly :: forall a. (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 5, g 4)
```

- What are the arities of f and g? Counting arrows...
 - $f :: \text{Int} \sim> \text{Int} \sim> a$ has arity 2
 - $g :: \text{Int} \sim> a$ has arity 1
- But what if $a = \text{Bool} \sim> \text{Bool}$?

The Problem With Polymorphism

And Static Compilation

`poly :: forall a. (Int ~> Int ~> a) ~> (a, a)`
`poly f = let g :: Int ~> a = f 3 in (g 5, g 4)`

- What are the arities of `f` and `g`? Counting arrows...
 - `f :: Int ~> Int ~> a` has arity 2
 - `g :: Int ~> a` has arity 1
- But what if `a = Bool ~> Bool`?
 - `f :: Int ~> Int ~> Bool ~> Bool` has arity 3...

The Problem With Polymorphism

And Static Compilation

```
poly :: forall a. (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 5, g 4)
```

- What are the arities of f and g? Counting arrows...
 - $f :: \text{Int} \sim> \text{Int} \sim> a$ has arity 2
 - $g :: \text{Int} \sim> a$ has arity 1
- But what if $a = \text{Bool} \sim> \text{Bool}$?
 - $f :: \text{Int} \sim> \text{Int} \sim> \text{Bool} \sim> \text{Bool}$ has arity 3...
 - $g :: \text{Int} \sim> \text{Bool} \sim> \text{Bool}$ has arity 2... oops...

The Problem With Polymorphism

And Static Compilation

`poly :: forall a. (Int ~> Int ~> a) ~> (a, a)`
`poly f = let g :: Int ~> a = f 3 in (g 5, g 4)`

- What are the arities of `f` and `g`? Counting arrows...
 - `f :: Int ~> Int ~> a` has arity 2
 - `g :: Int ~> a` has arity 1
- But what if `a = Bool ~> Bool`?
 - `f :: Int ~> Int ~> Bool ~> Bool` has arity 3...
 - `g :: Int ~> Bool ~> Bool` has arity 2... oops...
- How to statically compile? Is `'g 5'` a call? A partial application?

Nonuniform Representation

And Static Compilation

Nonuniform Representation

And Static Compilation

- Primitive types:

Nonuniform Representation

And Static Compilation

- Primitive types:
- **Int#, Float#, Char#,
Array#...**

Nonuniform Representation

And Static Compilation

- Primitive types:
- Int#, Float#, Char#, Array#...
- Unboxed

Nonuniform Representation

And Static Compilation

- Primitive types:
- Int#, Float#, Char#, Array#...
- Unboxed
- Efficient passing

Nonuniform Representation

- Primitive types:
- Int#, Float#, Char#, Array#...
- Unboxed
- Efficient passing

And Static Compilation

- Different sizes

Nonuniform Representation

- Primitive types:
- Int#, Float#, Char#, Array#...
- Unboxed
- Efficient passing

And Static Compilation

- Different sizes
- Different locations

Nonuniform Representation

- Primitive types:
- Int#, Float#, Char#, Array#...
- Unboxed
- Efficient passing

And Static Compilation

- Different sizes
- Different locations
- **Different levity**

Nonuniform Representation

- Primitive types:
- Int#, Float#, Char#,
Array#...

- Unboxed
- Efficient passing

revapp :: forall a b. a -> (a -> b) -> b
revapp x f = f x

And Static Compilation

- Different sizes
- Different locations
- Different levity

Nonuniform Representation

- Primitive types:
- Int#, Float#, Char#, Array#...

- Unboxed
- Efficient passing

And Static Compilation

- Different sizes
- Different locations
- Different levity

revapp :: forall a b. a -> (a -> b) -> b
revapp x f = f x

(++) :: [a] -> [a] -> [a]
plusFloat# :: Float# -> Float# -> Float#

Nonuniform Representation

- Primitive types:
- Int#, Float#, Char#, Array#...

- Unboxed
- Efficient passing

And Static Compilation

- Different sizes
- Different locations
- Different levity

revapp :: forall a b. a -> (a -> b) -> b
revapp x f = f x

(++) :: [a] -> [a] -> [a]
plusFloat# :: Float# -> Float# -> Float#

revapp [0..3] (++) [4..9] vs revapp 2.5 (plusFloat# 1.5)

A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

- All polymorphism is *uniform*

A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

- All polymorphism is *uniform*
 - 'α' is represented as a pointer

A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

- All polymorphism is *uniform*
 - 'α' is represented as a pointer
 - 'α' has arity 0

A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

- All polymorphism is *uniform*
 - 'α' is represented as a pointer
 - 'α' has arity 0
- Restriction on quantifiers for all $a :: k$

A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

- All polymorphism is *uniform*
 - 'α' is represented as a pointer
 - 'α' has arity 0
- Restriction on quantifiers for all $a :: k. \dots$
 - Special kinds for unboxed (#) and non-zero arity (~) types

A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

- All polymorphism is *uniform*
 - 'a' is represented as a pointer
 - 'a' has arity 0
- Restriction on quantifiers for all $a :: k. \dots$
 - Special kinds for unboxed (#) and non-zero arity (~) types
 - k may be \star or $\star \rightarrow \star$ but never # or ~

A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

- All polymorphism is *uniform*
 - 'a' is represented as a pointer
 - 'a' has arity 0
- Restriction on quantifiers for all $a :: k. \dots$
 - Special kinds for unboxed (#) and non-zero arity (~) types
 - k may be \star or $\star \rightarrow \star$ but never # or ~
- Draconian restriction is unsatisfactory

A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

- All polymorphism is *uniform*
 - 'a' is represented as a pointer
 - 'a' has arity 0
- Restriction on quantifiers `forall a :: k. ...`
 - Special kinds for unboxed (#) and non-zero arity (~) types
 - k may be `*` or `*->*` but never # or ~
- Draconian restriction is unsatisfactory
 - **Too restrictive:** Identical definitions/code repeated for different types (like `error :: String -> a`)

A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

- All polymorphism is *uniform*
 - 'a' is represented as a pointer
 - 'a' has arity 0
- Restriction on quantifiers `forall a::k. ...`
 - Special kinds for unboxed (#) and non-zero arity (~) types
 - k may be `*` or `*->*` but never # or ~
- Draconian restriction is unsatisfactory
 - **Too restrictive:** Identical definitions/code repeated for different types (like `error :: String -> a`)
 - **Incompatible with kind polymorphism:** `forall k::Kind. forall a::k. ???`

Representation Polymorphism

Kinds As Representations

Representation Polymorphism

Kinds As Representations

- Generalize $a :: \star$ to $a :: \text{TYPE } r$

Representation Polymorphism

Kinds As Representations

- Generalize $a :: \star$ to $a :: \text{TYPE } r$
 - $r :: \text{Rep}$ is the *representation* of a

Representation Polymorphism

Kinds As Representations

- Generalize $a :: \star$ to $a :: \text{TYPE } r$
 - $r :: \text{Rep}$ is the *representation* of a
 - $\star = \text{TYPE Ptr}$

Representation Polymorphism

Kinds As Representations

- Generalize $a :: \star$ to $a :: \text{TYPE } r$
 - $r :: \text{Rep}$ is the *representation* of a
 - $\star = \text{TYPE Ptr}$

$\text{revapp } x \ f = f \ x$

Representation Polymorphism

Kinds As Representations

- Generalize $a :: \star$ to $a :: \text{TYPE } r$
 - $r :: \text{Rep}$ is the *representation* of a
 - $\star = \text{TYPE Ptr}$

$\text{revapp } x \ f = f \ x$

$\text{revapp} :: \text{forall } (r1, r2 :: \text{Rep}) \ (a :: \text{TYPE } r1) \ (b :: \text{Type } r2).$
 $\quad a \rightarrow (a \rightarrow b) \rightarrow b$

Representation Polymorphism

Kinds As Representations

- Generalize $a :: \star$ to $a :: \text{TYPE } r$
 - $r :: \text{Rep}$ is the *representation* of a
 - $\star = \text{TYPE Ptr}$

revapp x $f = f$ x

revapp $:: \text{forall } (r1, r2 :: \text{Rep})$ ($a :: \text{TYPE } r1$) ($b :: \text{Type } r2$).
 $a \rightarrow (a \rightarrow b) \rightarrow b$

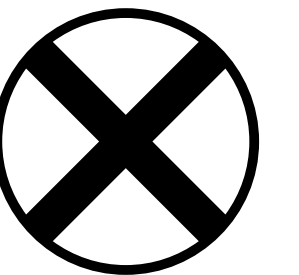
Representation Polymorphism

Kinds As Representations

- Generalize $a :: \star$ to $a :: \text{TYPE } r$
 - $r :: \text{Rep}$ is the *representation* of a
 - $\star = \text{TYPE } \text{Ptr}$

revapp x $f = f$ x

revapp $:: \text{forall } (r1, r2 :: \text{Rep}) (a :: \text{TYPE } r1) (b :: \text{Type } r2).$
 $a \rightarrow (a \rightarrow b) \rightarrow b$



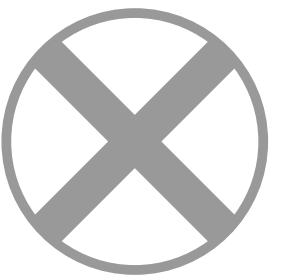
Representation Polymorphism

Kinds As Representations

- Generalize $a :: \star$ to $a :: \text{TYPE } r$
 - $r :: \text{Rep}$ is the *representation* of a
 - $\star = \text{TYPE Ptr}$

revapp $x \ f = f \ x$

revapp $:: \text{forall } (r1, r2 :: \text{Rep}) \ (a :: \text{TYPE } r1) \ (b :: \text{Type } r2).$
 $a \rightarrow (a \rightarrow b) \rightarrow b$



revapp $:: \text{forall } (r :: \text{Rep}) \ (a :: \text{TYPE } \text{Ptr}) \ (b :: \text{TYPE } r).$
 $a \rightarrow (a \rightarrow b) \rightarrow b$

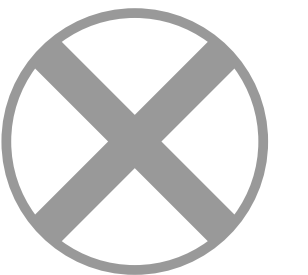
Representation Polymorphism

Kinds As Representations

- Generalize $a :: \star$ to $a :: \text{TYPE } r$
 - $r :: \text{Rep}$ is the *representation* of a
 - $\star = \text{TYPE Ptr}$

`revapp x f = f x`

`revapp :: forall (r1,r2::Rep) (a::TYPE r1) (b::Type r2).`
`a -> (a -> b) -> b`



`revapp :: forall (r::Rep) (a::TYPE Ptr) (b::TYPE r).`
`a -> (a -> b) -> b`

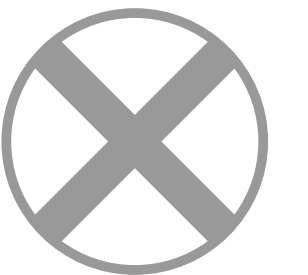
Representation Polymorphism

Kinds As Representations

- Generalize $a :: \star$ to $a :: \text{TYPE } r$
 - $r :: \text{Rep}$ is the *representation* of a
 - $\star = \text{TYPE Ptr}$

`revapp x f = f x`

`revapp :: forall (r1,r2::Rep) (a::TYPE r1) (b::Type r2).`
`a -> (a -> b) -> b`



`revapp :: forall (r::Rep) (a::TYPE Ptr) (b::TYPE r).`
`a -> (a -> b) -> b`

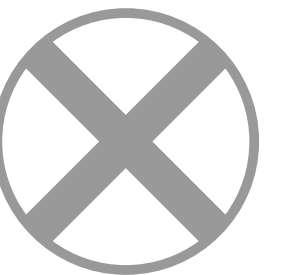
Representation Polymorphism

Kinds As Representations

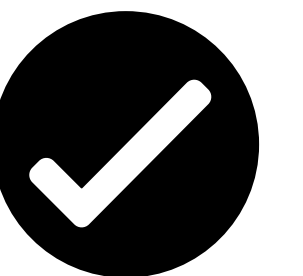
- Generalize $a :: \star$ to $a :: \text{TYPE } r$
 - $r :: \text{Rep}$ is the *representation* of a
 - $\star = \text{TYPE Ptr}$

revapp x f = f x

revapp :: forall (r1,r2::Rep) (a::TYPE r1) (b::Type r2).
a -> (a -> b) -> b



revapp :: forall (r::Rep) (a::TYPE Ptr) (b::TYPE r).
a -> (a -> b) -> b



Arity Polymorphism

Kinds As Calling Conventions

Arity Polymorphism

Kinds As Calling Conventions

- Generalize $a :: \text{TYPE } r$ to $a :: \text{TYPE } r \ c$

Arity Polymorphism

Kinds As Calling Conventions

- Generalize $a :: \text{TYPE } r$ to $a :: \text{TYPE } r \ c$
 - $r :: \text{Rep}$ is the *runtime representation* of a

Arity Polymorphism

Kinds As Calling Conventions

- Generalize $a :: \text{TYPE } r$ to $a :: \text{TYPE } r \ c$
 - $r :: \text{Rep}$ is the *runtime representation* of a
 - $c :: \text{Conv}$ is the *calling convention* of a

Arity Polymorphism

Kinds As Calling Conventions

- Generalize $a :: \text{TYPE } r$ to $a :: \text{TYPE } r \ c$
 - $r :: \text{Rep}$ is the *runtime representation* of a
 - $c :: \text{Conv}$ is the *calling convention* of a
 - $a :: \text{TYPE } \text{Ptr } \text{Call}[n]$ says values of a are pointers with arity n (simplified)

Arity Polymorphism

Kinds As Calling Conventions

- Generalize $a :: \text{TYPE } r$ to $a :: \text{TYPE } r \ c$
 - $r :: \text{Rep}$ is the *runtime representation* of a
 - $c :: \text{Conv}$ is the *calling convention* of a
 - $a :: \text{TYPE } \text{Ptr } \text{Call}[n]$ says values of a are pointers with arity n (simplified)

```
poly :: forall a :: TYPE Ptr Call[2]. (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)
```


Arity Polymorphism

Kinds As Calling Conventions

- Generalize $a :: \text{TYPE } r$ to $a :: \text{TYPE } r \ c$

- $r :: \text{Rep}$ is the *runtime representation* of a

- $c :: \text{Conv}$ is the *calling convention* of a

- $a :: \text{TYPE } \text{Ptr } \text{Call}[n]$ says values of a are pointers with arity n (simplified)

$\text{poly} :: \text{forall } a :: \text{TYPE } \text{Ptr } \text{Call}[2]. (\text{Int } \sim> \text{Int } \sim> a) \sim> (a, a)$

$\text{poly } f = \text{let } g :: \text{Int } \sim> a = f \ 3 \ \text{in } (g \ 4, g \ 5)$

- $f :: \text{Int } \sim> \text{Int } \sim> a :: \text{TYPE } \text{Ptr } \text{Call}[4]$ has arity 4 (2 + 1 + 1)

Arity Polymorphism

Kinds As Calling Conventions

- Generalize $a :: \text{TYPE } r$ to $a :: \text{TYPE } r \ c$

- $r :: \text{Rep}$ is the *runtime representation* of a
- $c :: \text{Conv}$ is the *calling convention* of a
- $a :: \text{TYPE } \text{Ptr } \text{Call}[n]$ says values of a are pointers with arity n (simplified)

$\text{poly} :: \text{forall } a :: \text{TYPE } \text{Ptr } \text{Call}[2]. (\text{Int } \sim> \text{Int } \sim> a) \sim> (a, a)$

$\text{poly } f = \text{let } g :: \text{Int } \sim> a = f \ 3 \ \text{in } (g \ 4, g \ 5)$

- $f :: \text{Int } \sim> \text{Int } \sim> a :: \text{TYPE } \text{Ptr } \text{Call}[4]$ has arity 4 (2 + 1 + 1)
- $g :: \text{Int } \sim> a :: \text{TYPE } \text{PTR } \text{Call}[3]$ has arity 3 (2 + 1)

Arity Polymorphism

Kinds As Calling Conventions

- Generalize $a :: \text{TYPE } r$ to $a :: \text{TYPE } r \ c$

- $r :: \text{Rep}$ is the *runtime representation* of a
- $c :: \text{Conv}$ is the *calling convention* of a
- $a :: \text{TYPE } \text{Ptr } \text{Call}[n]$ says values of a are pointers with arity n (simplified)

$\text{poly} :: \text{forall } a :: \text{TYPE } \text{Ptr } \text{Call}[2]. (\text{Int } \sim> \text{Int } \sim> a) \sim> (a, a)$

$\text{poly } f = \text{let } g :: \text{Int } \sim> a = f \ 3 \ \text{in } (g \ 4, g \ 5)$

- $f :: \text{Int } \sim> \text{Int } \sim> a :: \text{TYPE } \text{Ptr } \text{Call}[4]$ has arity 4 (2 + 1 + 1)
- $g :: \text{Int } \sim> a :: \text{TYPE } \text{Ptr } \text{Call}[3]$ has arity 3 (2 + 1)

$\text{revapp} :: \text{forall } (c :: \text{Conv}) (r :: \text{Rep})$
 $(a :: \text{TYPE } \text{Ptr } c) (b :: \text{TYPE } r \ \text{Call}[1]).$

$a \sim> (a \sim> b) \sim> b$

$\text{revapp } x \ f = f \ x$

Arity Polymorphism

Kinds As Calling Conventions

- Generalize $a :: \text{TYPE } r$ to $a :: \text{TYPE } r \ c$

- $r :: \text{Rep}$ is the *runtime representation* of a
- $c :: \text{Conv}$ is the *calling convention* of a
- $a :: \text{TYPE } \text{Ptr } \text{Call}[n]$ says values of a are pointers with arity n (simplified)

$\text{poly} :: \text{forall } a :: \text{TYPE } \text{Ptr } \text{Call}[2]. (\text{Int } \sim> \text{Int } \sim> a) \sim> (a, a)$

$\text{poly } f = \text{let } g :: \text{Int } \sim> a = f \ 3 \ \text{in } (g \ 4, g \ 5)$

- $f :: \text{Int } \sim> \text{Int } \sim> a :: \text{TYPE } \text{Ptr } \text{Call}[4]$ has arity 4 (2 + 1 + 1)
- $g :: \text{Int } \sim> a :: \text{TYPE } \text{Ptr } \text{Call}[3]$ has arity 3 (2 + 1)

$\text{revapp} :: \text{forall } (c :: \text{Conv}) (r :: \text{Rep})$
 $(a :: \text{TYPE } \text{Ptr } c) (b :: \text{TYPE } r \ \text{Call}[1]).$

$a \sim> (a \sim> b) \sim> b$

$\text{revapp } x \ f = f \ x$

- $f :: a \sim> b :: \text{TYPE } \text{Ptr } \text{Call}[2]$ has arity 2

Arity Polymorphism

Kinds As Calling Conventions

- Generalize $a :: \text{TYPE } r$ to $a :: \text{TYPE } r \ c$

- $r :: \text{Rep}$ is the *runtime representation* of a
- $c :: \text{Conv}$ is the *calling convention* of a
- $a :: \text{TYPE } \text{Ptr } \text{Call}[n]$ says values of a are pointers with arity n (simplified)

```
poly :: forall a :: TYPE Ptr Call[2]. (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 4, g 5)
```

- $f :: \text{Int } \sim\> \text{Int } \sim\> a :: \text{TYPE } \text{Ptr } \text{Call}[4]$ has arity 4 (2 + 1 + 1)
- $g :: \text{Int } \sim\> a :: \text{TYPE } \text{Ptr } \text{Call}[3]$ has arity 3 (2 + 1)

```
revapp :: forall (c :: Conv) (r :: Rep)
         (a :: TYPE Ptr c) (b :: TYPE r Call[1]).
         a ~> (a ~> b) ~> b
```

```
revapp x f = f x
```

- $f :: a \sim\> b :: \text{TYPE } \text{Ptr } \text{Call}[2]$ has arity 2
- $x :: a :: \text{TYPE } \text{Ptr } c$ is represented as a pointer

Levity Polymorphism

Call vs Eval, Revisited

Levity Polymorphism

Call vs Eval, Revisited

- Code that isn't **called** is **evaluated**

Levity Polymorphism

Call vs Eval, Revisited

- Code that isn't **called** is **evaluated**
 - `Eval U :: Conv` — eager (call-by-value) evaluation, Unlifted values

Levity Polymorphism

Call vs Eval, Revisited

- Code that isn't **called** is **evaluated**
 - `Eval U :: Conv` — eager (call-by-value) evaluation, Unlifted values
 - `Eval L :: Conv` — lazy (call-by-need) evaluation, Lifted values

Levity Polymorphism

Call vs Eval, Revisited

- Code that isn't **called** is **evaluated**
 - $\text{Eval } U :: \text{Conv}$ — eager (call-by-value) evaluation, Unlifted values
 - $\text{Eval } L :: \text{Conv}$ — lazy (call-by-need) evaluation, Lifted values
 - $\text{Eval } g :: \text{Conv}$ — polymorphic evaluation, with Levity variable g

Levity Polymorphism

Call vs Eval, Revisited

- Code that isn't **called** is **evaluated**
 - `Eval U :: Conv` — eager (call-by-value) evaluation, Unlifted values
 - `Eval L :: Conv` — lazy (call-by-need) evaluation, Lifted values
 - `Eval g :: Conv` — polymorphic evaluation, with `Levity` variable `g`

`Int g :: Type Ptr (Eval g) -- boxed, levity-g integers`

Levity Polymorphism

Call vs Eval, Revisited

- Code that isn't **called** is **evaluated**
 - `Eval U :: Conv` — eager (call-by-value) evaluation, Unlifted values
 - `Eval L :: Conv` — lazy (call-by-need) evaluation, Lifted values
 - `Eval g :: Conv` — polymorphic evaluation, with `Levity` variable `g`

```
Int g :: Type Ptr (Eval g) -- boxed, levity-g integers
```

```
sum :: forall (g1, g2 :: Levity). [Int g1] ~> Int g2
```

```
sum [] = 0
```

```
sum (x : xs) = x + sum xs
```

Levity Polymorphism

Call vs Eval, Revisited

- Code that isn't **called** is **evaluated**
 - Eval U :: Conv — eager (call-by-value) evaluation, Unlifted values
 - Eval L :: Conv — lazy (call-by-need) evaluation, Lifted values
 - Eval g :: Conv — polymorphic evaluation, with Levity variable g

```
Int g :: Type Ptr (Eval g) -- boxed, levity-g integers
```

```
sum :: forall (g1, g2 :: Levity). [Int g1] ~> Int g2
```

```
sum [] = 0
```

```
sum (x : xs) = x + sum xs
```

Levity Polymorphism

Call vs Eval, Revisited

- Code that isn't **called** is **evaluated**
 - Eval U :: Conv — eager (call-by-value) evaluation, Unlifted values
 - Eval L :: Conv — lazy (call-by-need) evaluation, Lifted values
 - Eval g :: Conv — polymorphic evaluation, with Levity variable g

```
Int g :: Type Ptr (Eval g) -- boxed, levity-g integers
```

```
sum :: forall (g1, g2 :: Levity). [Int g1] ~> Int g2
```

```
sum [] = 0
```

```
sum (x : xs) = x + sum xs
```

Levity Polymorphism

Call vs Eval, Revisited

- Code that isn't **called** is **evaluated**
 - Eval U :: Conv — eager (call-by-value) evaluation, Unlifted values
 - Eval L :: Conv — lazy (call-by-need) evaluation, Lifted values
 - Eval g :: Conv — polymorphic evaluation, with Levity variable g

```
Int g :: Type Ptr (Eval g) -- boxed, levity-g integers
```

```
sum :: forall (g1, g2 :: Levity). [Int g1] ~> Int g2
```

```
sum [] = 0
```

```
sum (x : xs) = x + sum xs
```

Levity Polymorphism

Call vs Eval, Revisited

- Code that isn't **called** is **evaluated**
 - Eval U :: Conv — eager (call-by-value) evaluation, Unlifted values
 - Eval L :: Conv — lazy (call-by-need) evaluation, Lifted values
 - Eval g :: Conv — polymorphic evaluation, with Levity variable g

```
Int g :: Type Ptr (Eval g) -- boxed, levity-g integers
```

```
sum :: forall (g1, g2 :: Levity). [Int g1] ~> Int g2
```

```
sum [] = 0
```

```
sum (x : xs) = x + sum xs
```


Levity Polymorphism

Call vs Eval, Revisited

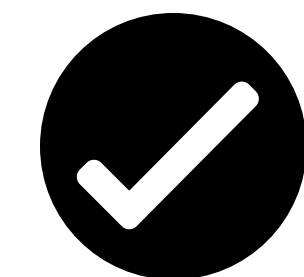
- Code that isn't **called** is **evaluated**
 - Eval U :: Conv — eager (call-by-value) evaluation, Unlifted values
 - Eval L :: Conv — lazy (call-by-need) evaluation, Lifted values
 - Eval g :: Conv — polymorphic evaluation, with Levity variable g

```
Int g :: Type Ptr (Eval g) -- boxed, levity-g integers
```

```
sum :: forall (g1, g2 :: Levity). [Int g1] ~> Int g2
```

```
sum [] = 0
```

```
sum (x : xs) = x + sum xs
```



Static Compilation

To the Machine

Static Compilation

To the Machine

- Only basic types (pointer, integer, float); no polymorphism

Static Compilation

To the Machine

- Only basic types (pointer, integer, float); no polymorphism
- Only fully saturated functions and calls

Static Compilation

To the Machine

- Only basic types (pointer, integer, float); no polymorphism
- Only fully saturated functions and calls

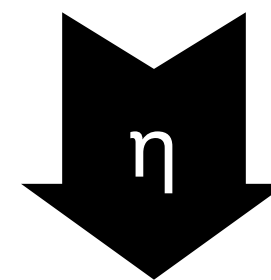
```
poly :: forall a :: TYPE Ptr Call[2].  
      (Int# ~> Int# ~> a) ~> (a, a)  
poly f = let g :: Int# ~> a = f 3  
         in (g 4, g 5)
```

Static Compilation

To the Machine

- Only basic types (pointer, integer, float); no polymorphism
- Only fully saturated functions and calls

```
poly :: forall a :: TYPE Ptr Call[2].  
      (Int# ~> Int# ~> a) ~> (a, a)  
poly f = let g :: Int# ~> a = f 3  
          in (g 4, g 5)
```

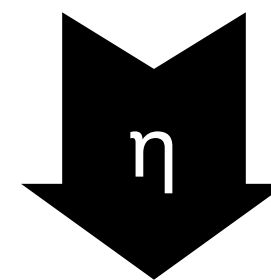


Static Compilation

To the Machine

- Only basic types (pointer, integer, float); no polymorphism
- Only fully saturated functions and calls

```
poly :: forall a :: TYPE Ptr Call[2].  
      (Int# ~> Int# ~> a) ~> (a, a)  
poly f = let g :: Int# ~> a = f 3  
          in (g 4, g 5)
```



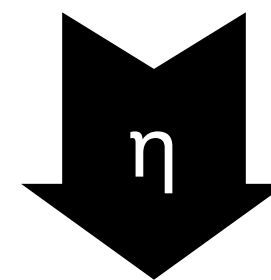
```
poly = \(f::Ptr) ->
```

Static Compilation

To the Machine

- Only basic types (pointer, integer, float); no polymorphism
- Only fully saturated functions and calls

```
poly :: forall a :: TYPE Ptr Call[2].  
      (Int# ~> Int# ~> a) ~> (a, a)  
poly f = let g :: Int# ~> a = f 3  
         in (g 4, g 5)
```



```
poly = \ (f :: Ptr) ->  
      let g :: Ptr = \ (x :: I32, y :: ?, z :: ?) -> f(3, x, y, z)
```

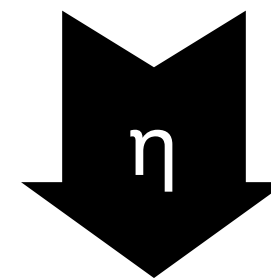

Static Compilation

With Polymorphic η -Expansion

Static Compilation

With Polymorphic η -Expansion

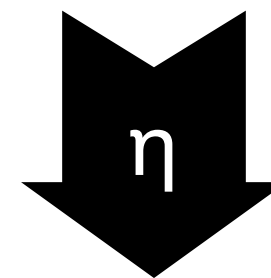
```
poly :: forall a::TYPE Ptr Call[Ptr, F64].  
      (Int# ~> Int# ~> a) ~> (a, a)  
poly f = let g :: Int# ~> a = f 3  
          in (g 4, g 5)
```



Static Compilation

With Polymorphic η -Expansion

```
poly :: forall a::TYPE Ptr Call[Ptr, F64].  
      (Int# ~> Int# ~> a) ~> (a, a)  
poly f = let g :: Int# ~> a = f 3  
          in (g 4, g 5)
```

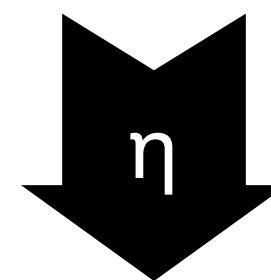


```
poly = \(f::Ptr) ->
```

Static Compilation

With Polymorphic η -Expansion

```
poly :: forall a::TYPE Ptr Call[Ptr, F64].  
      (Int# ~> Int# ~> a) ~> (a, a)  
poly f = let g :: Int# ~> a = f 3  
          in (g 4, g 5)
```

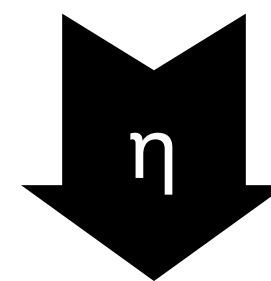


```
poly = \ (f::Ptr) ->  
      let g::Ptr = \ (x::I32, y::Ptr, z::F64) -> f(3, x, y, z)
```

Static Compilation

With Polymorphic η -Expansion

```
poly :: forall a::TYPE Ptr Call[Ptr, F64].  
      (Int# ~> Int# ~> a) ~> (a, a)  
poly f = let g :: Int# ~> a = f 3  
          in (g 4, g 5)
```



```
poly = \ (f::Ptr) ->  
      let g::Ptr = \ (x::I32, y::Ptr, z::F64) -> f(3,x,y,z)  
      in (\ (y::Ptr, z::F64) -> g(4, y, z),  
         \ (y::Ptr, z::F64) -> g(5, y, z))
```

Even More

In the Paper

- **Levity Polymorphism**
 - For when evaluation strategy doesn't matter
- **Compiling Source \rightarrow Intermediate \rightarrow Target**
 - Via kind-directed η -expansion and register assignment
- **Type system for ensuring static compilation**
 - Of definitions with arity, levity, and representation polymorphism

Kinds capture the details of
efficient calling conventions in
low-level machine code